UNIX EMULATION SERVICES FOR PAWAN (II)

By

Deepankar Bairagi

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1993

UNIX EMULATION SERVICES FOR PAWAN (II)

A thesis submitted
in partial fulfillment
of the requirements
for the degree of

Master of Technology

by

Deepankar Bairagi

to the

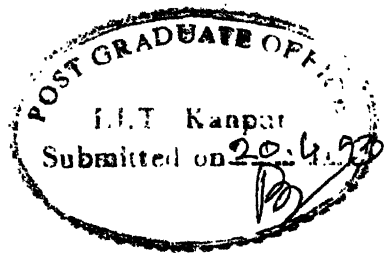
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

April, 1993

CSE – 1998 – M.
BAI – U

CS

CERTIFICATE



This is to certify that the work contained in the thesis titled, **UNIX Emulation Services for PAWAN (II)**, was carried out under my supervision by **Deepankar Bairagi** and it has not been submitted elsewhere for a degree.

Gautam Barua
Professor
Dept. of Comp. Sc. and Engg.
I.I.T., Kanpur
April 1993

CSE-1993-M-BAI-UNI

To My Parents

ACKNOWLEDGEMENT

I am grateful to my thesis supervisor Dr Gautam Barua for his invaluable guidance and constant encouragement throughout the course of this thesis. I would like to thank Hazel for being so cooperative, our seniors in the MACH group without whose help this project would have been an uphill task and all the nightbirds in CC for keeping company.

I thank all my friends and classmates who have made my stay here really enjoyable. My thanks to the CSE lab staff, particularly Mr Pandit, for being so helpful. And last but not the least, I acknowledge His Divine Grace.

DEEPANKAR BAIRAGI

April, 1993.

Abstract

In this thesis the design and implementation of a UNIX emulation service for PAWAN is described. The trend in operating systems research in the last few years has been shifted from kernalization to dekernalization i.e kernels enriched and overloaded with functionality and abstractions are steadily making way for micro-kernels. The design and implementation of PAWAN is based on MACH, a multiprocessor based micro-kernel developed at Carnegie Mellon University. The MACH micro-kernel was ported onto the M68020 based Horizon III and a set of user state servers was added to it by a group of four students. This thesis is a continuation of their work.

PAWAN consists of an emulation library and a set of user state servers which successfully exploit the MACH features to emulate UNIX. The user state servers provide file service, signalling facility, terminal handling, naming, process control and program execution. In the present status, complete UNIX source code compatibilty has been achieved. By porting sophisticated utilities with minimal effort, a fullfledged operating system can be built on the top of it.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The enhancements in hardware technology, experience with existing systems, development of new applications and needs have resulted in dramatic changes in operating systems. The user has become literally swamped with systems offering newer and better facilities. Under the weight of changing needs and technologies, operating systems have been modified to provide a staggering number of different mechanisms for management of objects and resources. These new features have mostly been incorporated into the kernel, making it unwieldy and reducing modifiability.

The advent of high speed networks and the demand for integrated distributed systems has coincided with the development of better structures for conventional operating systems, allowing them to be built as a collection of intercommunicating software modules or processes each performing a well-defined function, such as file management resource allocation or task scheduling. The advent of easily configurable open systems to which new services can be added without having to rebuild or restart existing system software, has been a positive step in this direction. In such systems the boundary between the operating systems and the programs developed by users is less rigid, allowing the system to be viewed as an extensible set of software resources and services. Examples of such systems are the MACH, AMOEBA, V-system, CHORUS and the LOCUS distributed systems.

The MACH technology has been designed for building the "new generation" of open, distributed scalable operating systems. The MACH kernel abstractions essentially provides a base upon which complete system environments may be built. At IIT Kanpur, the MACH microkernel has been ported to run on a network of MC68020 based uniprocessors and several user state servers has been implemented [Ashi92, Baru92, Gopa92, Rao92] in order to handle most of the functionality of the UNIX system. This and its companion thesis [Das93] describes the enhancements to these services and the construction of an emulation library to provide complete UNIX source code compatibility.

1.1 MACH Operating system

The MACH micro-kernel developed at Carnegie Mellon University is a successor to the ACCENT project and is intended as a basis for development of distributed systems that include multiprocessors. It is an open system based on a lightweight kernel running in each computer with services such as file system, network services and process management outside the kernel [Coul88].

MACH is fundamentally a message passing communication kernel. Operations on objects other than messages are performed by sending messages to ports. In this way MACH permits system services and resources to be managed by user-state tasks.

The model provided by MACH is a service model in which objects are managed by servers and clients make requests for operation on objects by using remote procedure calls. The Matchmaker remote procedure call interface language [Jone86] enables distributed programs to be built in several existing programming languages including C, Pascal, Ada and Lisp. Remote procedure calling in Matchmaker is supported by efficient and flexible interprocess communication facilities in the kernel.

MACH objects are represented by ports, which are protected message queues, and whose names can be transmitted in messages. By using Matchmaker, application programmers can be shielded from the intricacies of message composition, sending and reception and instead can be offered a procedural interface for sets of typed operations upon objects.

The MACH virtual memory management system exhibits architecture independence, multiprocessor and distributed system support and advanced functionality.

1.2 Overview of Other Distributed Systems

1.2.1 The V Distributed system

The V distributed system [Cher84], is an operating system designed for a cluster of computer workstations connected by a high performance network. The system is structured as a relatively small "distributed kernel", a set of service modules, various run-time libraries and a set commands. The kernel is distributed in that a separate copy of the kernel executes on each participating network node, yet separate copies cooperate to provide a single system abstraction of processes in address spaces communicating using a base set of communication primitives.

The existence of multiple machines and network interconnections is largely transparent at process level. The service modules implement value-added

services using the basic access to hardware resources provided by the kernel. The various run-time libraries implement conventional language or application-to-operating system interfaces.

The kernel is a software backplane that provides a base for building and configuring systems. It consists of lightweight processes and interprocess communication. New services are 'plugged in' and communicate with other processes using the IPC provided in the kernel and their internal design is invisible to the rest of the system. The V-kernel runs in each workstation and is based on clusters (V-teams) of threads, that can be dynamically created and destroyed and share an address space. Processes communicate via shared memory within a V-team and via network transparent, synchronous message passing between different V-teams.

1.2.2 AMOEBA

The AMOEBA architecture consists of four principal components [Rene89], [Mull85], [Coul88]. First are the workstations, one per user, which run window management software and on which users can carry out tasks requiring fast interactive response. Then we have the pool processors, a group of CPUs that can be dynamically allocated as needed, used and then returned to the pool. The specialised servers such as directory, file server and various servers provide specialised functions. Fourth are the wide area network gateways, which are used to link AMOEBA systems at different sites in possibly different countries into a single uniform system.

AMOEBA is an object oriented distributed system, the objects are abstract data types such as files, directories and processes and are managed by server processes. A client process carries out operations on an object (transaction), by sending a request message to the server process that manages the object. While the client blocks, the server performs the requested operation on the object and sends a reply message back to the client which unblocks the client. To handle multiple transactions going on at the same time a process can be subdivided into lightweight subprocesses called threads. By having a thread for each request, a server process can handle multiple requests simultaneously. A client process can perform several transactions at the same time by having a thread per transaction.

1.2.3 The CHORUS Distributed System

The CHORUS distributed system enables us to integrate various types of operating systems - from small real-time systems to general-purpose operating systems, in a single operation system [Abro89],[Coul88]. CHORUS is a communication based technology. Its minimal real-time kernel integrates distributed processing and communications at the lowest level.

CHORUS operating systems are built as sets of independent system servers, to which the kernel provides the basic services such as activity

scheduling, network transparent IPC, memory management and real-time event handling. The kernel can be scaled to exploit a wide range of hardware configurations such a small embedded board, multiprocessor workstations or high performance server. The memory management service has been designed as a well isolated component offering generic interfaces adapted to various hardware architectures and to various system needs. The secondary storage objects are managed outside the kernel, within independent servers.

An actor is the unit of distribution in the CHORUS system. It defines a protected address space supporting the execution of threads which share the address space of the actor.

The thread is the unit of execution in a CHORUS system, it is characterised by a context corresponding to the state of the processor. A thread is always tied to one and only one actor, which constitutes the execution environment of the thread. Within the actor many threads can be created and can run in parallel. Threads making up an actor can communicate and synchronise with any other thread, on any site. The CHORUS virtual memory has been designed as a set of basic tools suited for versatile implementation of various system policies.

1.2.4 The X-kernel

The X-kernel is an experimental operating system for personal workstations that allow uniform access to resources throughout a nationwide internet: an interconnection of networks similar to the TCP/IP internet [Pete90]. Workstations are viewed as a portal through which users access both local and remote network resources. This is realised by providing an infrastructure that is general enough to support a wide variety of protocols, yet efficient enough that no protocol suffers a serious performance penalty.

The X-kernel supports a library of protocols, and it accesses different resources with different protocol combinations. Two user-level systems have been built on top of the X-kernel to give users an integrated and uniform interface to resources. These systems, a file system and a command interpreter hide differences among the underlying protocol.

The X-kernel currently runs on the SUN-3 workstations and incorporates components that manage processes, memory and communication. Multiple address spaces are supported, multiple light-weight processes can execute in each address space and processes within an address space synchronise using kernel supported semaphore. A communication manager is provided, which offers an object oriented infrastructure for composing protocols and a collection of powerful tools for implementing tasks common to all protocols.

1.2.5 The LOCUS Distributed System

LOCUS is an UNIX-like distributed system [Walk83],[Coul88]. It supports transparent access to data through a network wide file system, permits au-

tomatic replication of storage, supports transparent distributed process execution, supplies a number of high reliability functions such as nested transactions, and is upward compatible with UNIX. The system provides a high degree of transparency concerning the location of files and some degree of transparency concerning the location of process execution.

The system appears to clients like one giant UNIX system, with all the computers playing both client and server roles and with UNIX file access, process creation and interprocess communication primitives implemented transparently across the network. LOCUS is a procedure based operating system, processes request system services by executing system calls, which trap to the kernel.

The LOCUS filesystem presents a single tree structured naming hierarchy to applications and users. LOCUS names are fully transparent; it is not possible from the name of the resource to discern its location in the network. LOCUS provides nested transactions in which all changes to a given file are atomic. To implement transactions, both original and changed data are kept at the storage site for the file in question, until a transaction is complete. LOCUS provides replication of files at the granularity of whole directories on the grounds that, if some node in a tree is inaccessible, then all the files below that node are inaccessible.

1.3 Motivation

Continuously evolving microprocessors and the broad spectrum of new software have made configurability and modifiability, even at the cost of a little inefficiency, important factors to look for in a new system. MACH provides a powerful standard micro-kernel, which can eventually run different operating system applications side by side on the same workstation.

We selected MACH for our purpose from the other alternative systems described above. LOCUS tries to provide a distributed environment for UNIX, resulting in a closed system with with a very large kernel [Coul88]. A extensive protocol library has been implemented in X-kernel to provide uniform access to resources in a nationwide internet. All the others have implemented lightweight kernels as a basis for building distributed systems. MACH provides multiple tasks, with multiple threads of execution and integrates both multiprocessor functionality. It has the technology to deliver a high performance inter-process communication. MACH is also exploring the extension of memory from the local machine to a transparent network-wide service [Gold89]. MACH is available and supported as a product by a number of hardware vendors, including NEXT. It is the base technology for the OSF/1 operating system from the Open Software foundation . MACH has become the most widely used micro-kernel not only for UNIX based operating systems but even as the core for the next generation of IBM's OS/2 (UNIX WORLD oct 1992).

The source code for MACH3.0 micro-kernel had been ported to the MC68020 based mini, as the basis for a distributed OS to act as an workbench for experimental distributed operating systems. Various distributed services has been provided by building a number of user-state servers. We have augmented the user-state servers and built an emulator for UNIX on top of the MACH kernel.

In this thesis and the companion thesis [Das93], we present an overview and discussion of the major techniques and experiments that characterise the design of the emulation system. Some of the relevant aspects include the combination of several independent servers to create a complete system, generic service interfaces relying on the emulation library as a interface translator, and the moving of portions of system state and processing from servers into an emulation library.

1.4 Outline of Thesis

The Key ingredients of the MACH kernel and its outstanding characteristics are dealt with in the next chapter. Chapter 3 explores the PAWAN design and describes the UNIX emulation setup. The design goals and implementation details of the File server are given in chapter 4. The performance of PAWAN is analysed in Chapter 5. In Chapter 6, we have explored the possible server extensions and improvements. Details of the Proccess Server, Exec Server and the Emulation Library can be got from [Das93]. The other user state servers are explained in [Ashi92], [Baru92] [Gopa92], [Rao92].

Chapter 2

MACH Overview

This chapter describes MACH and the motivations that led to its design. The kernel abstractions and the features of MACH are examined. The basic abstractions and functions ported to the IITK implementation are also briefly discussed.

2.1 The MACH micro-kernel architecture

The recent trend in operating system development consists of structuring the operating system as a modular set of system servers, which sit on top of a minimal micro-kernel. Examples of such systems have been given in chapter 1 (MACH, CHORUS, AMOEBA and V-system). The micro-kernel provides system servers with generic tools, limited to process scheduling and memory management functions, independent of a particular operation system environment, and a simple Inter-Process communication(IPC) facility that allows system servers to interact independently of where they are executed, in a multiprocessor, multicomputer, or network configuration [Gien91].

The MACH micro-kernel has been selected as the base for our emulation system for UNIX. The basic kernel services forms a standard base supporting the implementation of functions specific to a particular operating system environment. These system specific functions can be configured into user-state servers managing the other physical and logical resources of a computer system such as files, devices and high level communication services. This increases the modularity, portability, scalability and "distributability" of the overall emulated system. We can build and design emulation systems for a range of targets that span from relatively simple systems like MS-DOS to considerably larger systems such as VMS [Dean90],[Juli92]. The details of our UNIX emulation system is given in chapter 3.

2.2 Design: an extensible kernel

The MACH kernel abstractions provide a base upon which complete system environments may be built [Acce86], [Baro88], [Rash86]. MACH provides for the manipulation of system resources through this small set of machine-independent abstractions and for the integration of memory management and communication functions [Dean 90]. The MACH abstractions were chosen not only because of their simplicity but for performance reasons. Substantial performance benefits are gained by integrating virtual memory management and interprocess communication [Rash86], [Youn87]. The MACH kernel provides five basic abstractions.

- A *task* is an execution environment in which threads may run. It is the basic unit of resource allocation and includes a paged virtual address space and protected access to system resources such as processes and port. It is the framework in which a number of threads carry out computations.
- A *thread* is the basic unit of execution. It consists of a basic processor state, an execution stack and a limited amount of per thread static storage. It shares all other memory and resources with the other threads executing in the same task. It can execute only in one task.
- A *port* is a communication channel - logically a queue for messages protected by the kernel. Ports are the reference objects of the MACH design. Only one task can receive messages from a port, but all of tasks that have access to the port can send messages.
- A *message* is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain pointers and type capabilities for ports.
- A *memory object* is a secondary storage object that is mapped into a task's virtual memory. These are commonly files managed by a file server.

The MACH kernel functions can be divided into the following categories.

- basic message primitives and support facilities.
- port and port set management facilities.
- task and thread creation and management facilities.
- virtual memory management functions.
- operation on memory objects.

The provision of these basic functions in the kernel, makes it possible to run varying system configurations on different classes of machines while providing a consistent interface to all resources. The actual system running on any particular machine thus becomes a function of its server rather than its kernel.

2.2.1 Tasks and threads

MACH divides the UNIX process abstraction into two orthogonal abstractions : the *task* and the *thread*. MACH allows multiple threads to execute within a single task. On tightly coupled shared memory multiprocessors, multiple threads within the same task may execute in parallel. Thus an application can use the full parallelism available and still incur modest kernel overhead.

Operations on threads and tasks are invoked by sending a message to a port representing a task or thread. Threads may be created (within a specified task), destroyed, suspended and resumed. The suspend and resume operations, when applied to a task, affect all threads within the task.

Tasks are related to one another in a tree structure by task creation operation. Regions of virtual memory may be marked for future child tasks as either inheritable, read/write, copy- on-write, or neither.

Application parallelism can be achieved in MACH in three ways

- through creation of a single task with many threads of control, able to execute in a shared address space, using shared memory for communication and synchronization.
- through related creation of many tasks that share restricted regions of memory.
- through the creation of many tasks that communicate via messages.

2.2.2 Virtual memory management

The MACH virtual memory design allows tasks to

- allocate and deallocate regions of virtual memory.
- set protections on regions of virtual memory.
- specify the inheritance of regions of virtual memory.

It allows for both copy on write and read/write sharing of memory between tasks. Virtual memory related functions, such as pagein and pageout, may be performed by non-kernel tasks [Youn87],. MACH does not impose restrictions on what regions may be specified for these operations, except that they be aligned on system page boundaries.

An important feature of MACH's virtual memory is the ability to handle page faults and pageout data requests outside the kernel. When virtual memory is created, special paging tasks may be specified to handle paging requests. Thus to implement memory mapped files, virtual memory is created with its pager specified as the file system. MACH provides some basic paging services in the kernel. Memory with no pages is automatically zero-filled.

MACH implements virtual memory by mapping process addresses onto memory objects which are represented as communication channels and accessed via messages. This increases the flexibility in memory management available to user programs and improves performance. The memory object can be created and serviced by a user-level data manager task. This gives MACH the ability to efficiently manage system services like network paging and file system support outside the kernel.

MACH's virtual memory implementation is split between the machine-*independent* and the machine-*dependent* sections. This contributes to portability and supports very large but sparsely populated virtual memory spaces. The MACH address map need only contain descriptors for those regions of virtual memory actually occupied by useful items. In addition to the normal demand paging of tasks, MACH virtual memory implementation allows portions of the kernel to be paged.

2.2.3 Interprocess communication

The MACH interprocess communication is defined in terms of ports and messages and provides both location independence, security and data type tagging.

The basic transport abstraction provided by MACH is the *port*. A port is a protected kernel object onto which messages may be placed and from which messages may be removed. It may have any number of senders, but only one receiver. Access to port is granted by receiving a messagge containing a port capability (to either send or receive).

Interprocess interfaces, including the interface to MACH kernel, are defined using an interface definition language called matchmaker. Matchmaker allows a program to specify an interface between client and server. It then locates the specification and generates stubs to create a distributed program without worrying about details of sending messages or type conversion between various machines.

2.3 MACH tools

2.3.1 MIG - the MACH interface generator

MIG is an interim implementation of a subset of the Matchmaker language that generates C and C++ remote procedure call interfaces for interprocess communication between MACH tasks [Drav89]. The MIG program automatically generates procedures in C to pack and send, or receive and unpack the IPC messages used to communicate between processes.

The user must provide a specification file defining parameters of both the message passing interface and the procedure call interface. MIG then generates three files:

- User Interface Module: This module is meant to be linked into the client program. It implements and exports procedures and functions to send and receive the appropriate messages to and from the server.
- User Header Module: This module is meant to be included in the client code to define the types and routines needed at compilation time.
- Server Interface Module: This module is linked into the server process. It extracts the input parameters from an IPC message, and calls a server procedure to perform the operation. When the server procedure or function returns, the generated interface module gathers the output parameters and correctly formats a reply message.

MIG is implemented as a cover program that recognizes a few switches and then calls cpp to process comments and preprocessor macros such as #include or #define. The output from cpp is then passed to the program migcom which generates the C files. The server procedures must be written by the user. The server waits for procedure call invocations on its communication port. To use the calls exported by the user interface modulem a client must first access the port to call the server on.

2.3.2 C threads

The C Threads package allows parallel programming in C under the MACH operating system [Coop88], [Golu87]. The MACH kernel does not enforce a synchronisation model, it just provides basic primitives upon which different models of synchronisation may be built. The C-threads provides a high level C interface to the low level thread primitives along with a collection of other mechanisms useful in various parallel programming paradigms. It provides

- multiple threads of control for parallelism.
- shared variables.
- mutual exclusion for critical sections.

- condition variables for synchronization of threads.

The C-threads package in MACH has three possible implementations. The first implements threads as coroutines in a single task, the second uses a separate task for each C-thread, using inherited shared memory to partially simulate the environment in which multiple threads run. The third implements C-threads using MACH threads. IITK MACH provides the third implementation.

2.4 MACH I/O structure

The MACH I/O configuration is different from that of the current systems. The peripheral devices are accessed through the device server, which is implemented as a kernel object. Operations on the device server are invoked through request messages sent to the port representing the device server.

We have three device drivers, the terminal driver, the disk driver and the network driver.Device switch `dev_name_list` describes the devices possibly attached to the system. The table entries describes the entry points to the driver. We have yet another table, which gives the bus specification of devices. Character and block devices reside on the same table. MACH devices are accessed through block and character device interfaces to the kernel.

2.4.1 I/O processing

To perform I/O on a device, a device_open request is made to the device server, which allocates a port for the device. This port can be used to perform I/O on that device. We can have a trusted server performing access mediation for the device, but handing out the port to various tasks at its discretion. Send rights to the device server port implies complete control over all the devices, whereas access to a device port imply control over that particular device.

Normal I/O is done by sending request messages to the device port. I/O can be inband or out of band as suitable for specific devices. Data is not buffered by the device server to avoid hardwiring the policies within it. The applications which uses its services are expected to employ their own buffering schemes.

A special entry point is provided in the device switch to handle asynchronous input. This point `device_set_filter`, associates a filter with a port. The input passes through this filter before being queued at the specified port. This facility is in fact used by the TTY server to handle special characters.

2.5 The kernel file system and network support

The MACH kernel provides a basic file system, the boot_ufs (derived from the Berkeley Fast File system) within the kernel. It is made up of blocks of at most 8K units, with the smaller units (fragments) only in the last direct block. The fragment sizes are multiples of the device block. The inode is the focus of all file activity in the system. A unique inode is allocated for each active file, current directory , mount files, text file and the root.

By itself, the Mach kernel does not provide any mechanisms to support inter-process communication over the network. However, the definition of Mach IPC allows for communication to be transparently extended by user-level tasks called Network Servers. The BSD UNIX socket facility for network communication [Leff89], has been implemented inside the IITK MACH kernel [Baru91]. The user state network server can be written over this socket facility.

Chapter 3

The Design of PAWAN

In this chapter, we discuss the design goals of PAWAN, the emulation services, the user state servers in PAWAN and some issues regarding integration of the servers, user interface, authentication and system initialization etc.

3.1 Design Goals

Defining and standardizing a powerful micro-kernel base is only the first step in realizing the potential of the overall MACH approach. The next step, and perhaps the one richest in design possibilities, is to learn how to construct a wide range of useful higher-level systems on the top of this simple kernel. A particular class of such systems is the so-called emulation systems, that implement the application programming interface of an existing complete operating system or target system with various combinations of user level components viz. servers and/or libraries operating on the top of a MACH kernel. The main benefits expected from this overall emuation approach include increased modularity, portability, flexibility, security and extensibility, as well as simpler development, debugging and maintenance.[Juli92]

A major goal of PAWAN is to implement UNIX as an application program or to be specific, as a user task. Beyond the obvious advantages common to all client/server models, treating Unix as an application program has a number of implications :

- Tailorability : versions of UNIX such as 4.3 BSD, System V.4 can be treated as different applications which can be potentially run side by side.
- Portability : a considerable portion of UNIX code is not machine dependent.
- Extensibility : new versions of UNIX can be implemented or tested alongside existing versions.

- real-time : traditional barriers to real-time support in UNIX can be removed both because the kernel itself does not have to hold interrupt locks for a long period of time to accomodate UNIX services and because the UNIX services themselves are preemptable. [Dean90]

There are two basic approaches to this design :

1) Multi-threaded User-state Unix server: Such a design is shown in Fig 3.1. This idea has been implemented in CMU. A single server, implemented as a MACH task with multiple threads of control managed by the C threads package, provides the UNIX semantics. Its key components are :

- A transparent UNIX support library:
  - Interprets all UNIX traps
  - Some simple traps are handled locally
  - Other traps are translated into messages to server.
- Unix Server:
  - Derived from original BSD implementation
  - pageable, interruptible, multithreaded
  - Acts as memory object pager for Unix inodes

2) Multi-server user state Unix : The key components of such a system are:

- A transparent UNIX support library:
  - Interprets all UNIX traps
  - Some simple traps are handled locally
  - Other traps are translated into messages to server.
- Generic (non-UNIX) servers:
  - Name server, Authentication server etc.
- User specific servers
  - File server, pipe server etc.

Another approach might be a compromise between the above two i.e. having one main central server and a number of auxiliary servers implementing specific portions of functionality.

PAWAN has an emulation library and several user state servers which provide UNIX services in a completely transparent manner. The choice of this design has been largely determined by extrapolating from the architecture of the native implementation of the target operating system UNIX. In

a shared memory multiprocessor machine, a single monolithic UNIX server implemented through system call and exception redirection might prove to be more efficient [Acce86], [Teva87d], [Youn87]. But in a loosely coupled LAN based environment as ours, a centralized UNIX server which services every user request, system call or exception in the network will clearly be extremely inefficient[Ashi92][Baru92][Gopa92][Rao92]. For reasons stated above, the second approach has been chosen.

A major issue involved here is the separation of the service layer to two separate layers, one for generic and one for target specific functions. The target specific layer can typically be an emulation library. Generic servers can provide union of services provided by many different target systems. But generic interfaces are prohibitively complex and expensive although such a design will facilitate emulation of more than one operating system. Therefore, at the most one may afford to have only a partially generic service layer. The major driving factor behind the design of PAWAN is UNIX emulation and more particularly UNIX source code compatibility.

One might point out that in such systems, each individual component i.e. a server or a library is typically designed specifically for the particular system in which it is to be used, and implemented either from scratch or by adapting code from the target OS implementation. Therefore, a major goal has been to make use of the existing unix code wherever possible (File server and Network server).

In order to maximize the overall flexibility, it is desirable to have as many independent servers as possible. These servers can then be used as a collection of standardized building blocks that can be assembled in various ways to create different systems. In addition to flexibility, such an architecture also increases the security and robustness of the system by isolating faulty or potentially malicious components into separate protected address spaces. Moreover it sometimes simplifies the implementation of some servers. However, such a desire for maximum modularity must be balanced against a number of practical considerations [Juli92]. For example, interaction between servers are more expensive than interaction between modules inside a particular server. The separation of services and corresponding interfaces must be carefully defined to minimize those interactions.

Keeping these issues in view, UNIX services in PAWAN has been divided into six user-state servers, each of which is accompanied by a library. The servers are Exec server, File server, Process server, Environment server, TTY server and Network server. The overall design of pawan is given in Fig 3.2. Each emulated process is a separate MACH task that contains the unmodified user code from various application programs and an emulation library that intercepts and implements system calls issued by instructions embedded in the user code. A brief description of the servers is given in the next section.

3.2 Description of the Servers

3.2.1 The Process Server

The full functionality of the UNIX signalling facility is provided by PAWAN. The process server acts as the forwarding agent for all the 'kill' requests. We go for a server based implementation since we need access to the task kernel ports. The SIGKILL type of signals which can't be blocked, caught or ignored are taken care of by the Signal server. A task which handles signals will have a thread waiting for signals on its exception port. Since the process server contains entries by all the user task, the default action can be taken by the process server.

The Authentication server is implemented as a privileged thread within the process server. This turns out to be quite efficient since the process server tables contains information about all user processes(tasks).

3.2.2 The Exec Server

PAWAN provides UNIX style execve() as well as run() (a new call, details in the companion thesis[Das93]). These are implemented as emulation library routines. However the implementation of setid exec is not possible at this level. We need to update the privileges ,and this is possible only by the trusted servers.

We run the Exec server with root privileges at startup time to take care of the setid exec. The execve() and run() library routines detect the need and contacts the Exec server when a setid exec or run() comes up. The Exec Server then takes up the responsibility of updating the privileges and setting up the process.

3.2.3 The File Server

The file server in PAWAN is completely compatible with 4.3 BSD. It provides UNIX-like file I/O and can be integrated with the network server to provide transparent network-wide file access and shared memory [Rao92]. It has been implemented by porting the 4.3 BSD file system code and uses the device server inside the kernel for device I/O. The major difference of the file server with the BSD approach is the maintenance of U area inside the file server itself. It uses the notion of threads to exploit concurrency and uses features like copy-on-write to minimize copies of data. The File Server is described in detail in Chapter 4.

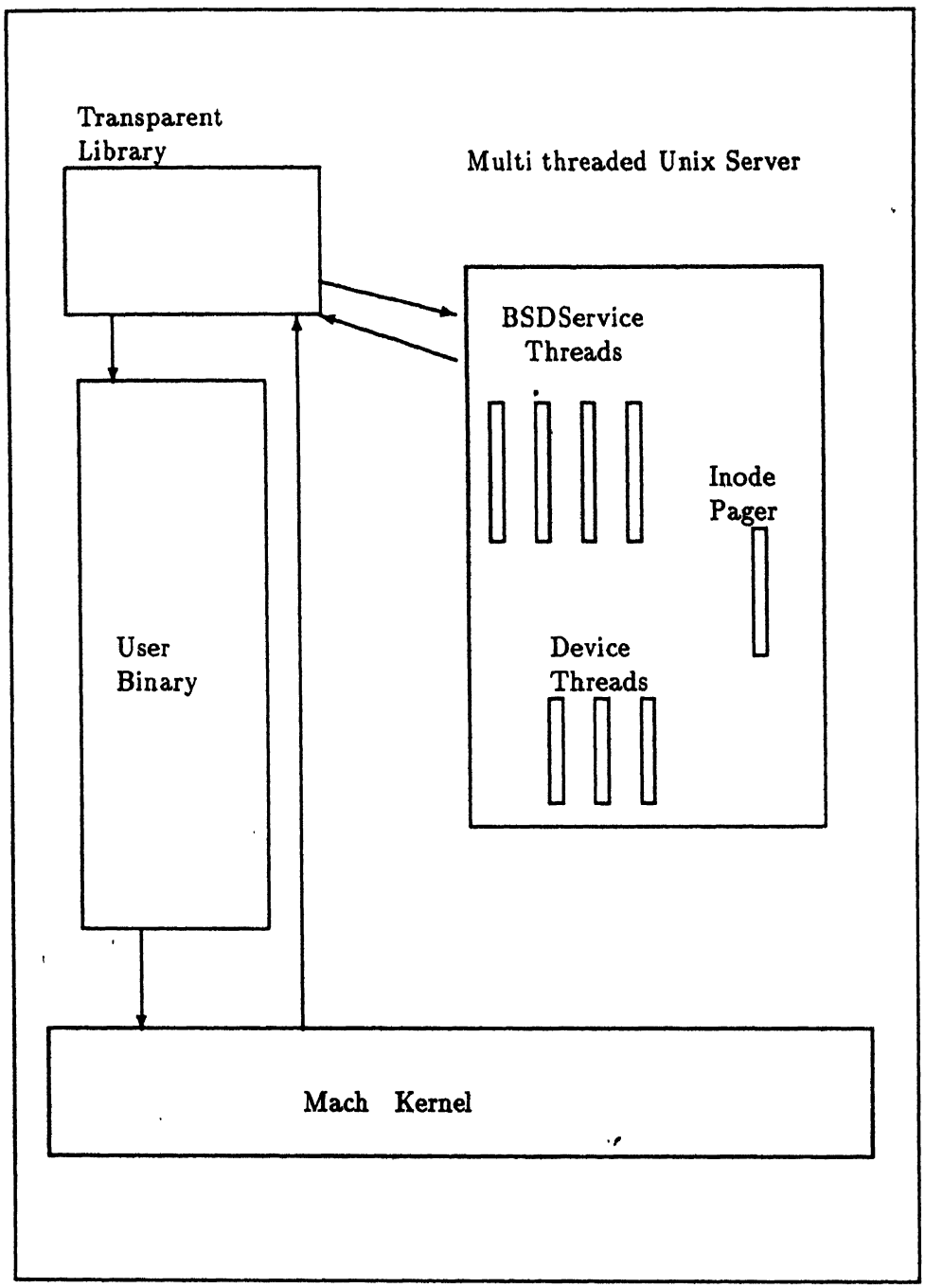


Figure 3.1: The Single Server Model

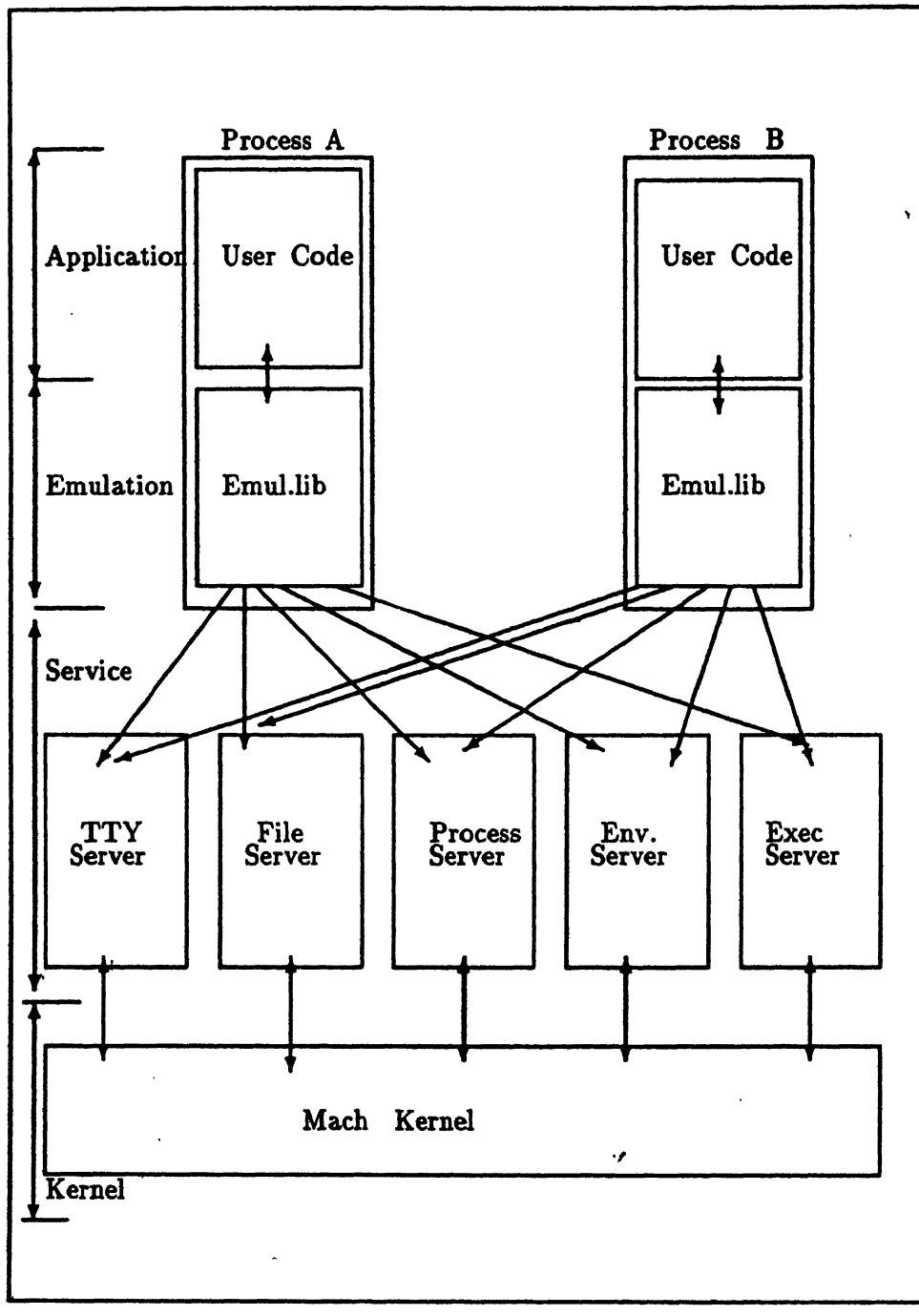


Figure 3.2: The design of PAWAN

3.2.4 The Environment Server

The environment server facilitates the sharing of named variables between tasks. It is an extension of the MACH EnvManager. An environment is a set of named variables which can be read or changed via calls on an Environment Port. Variables can be strings, ports or environments. An environment may be shared between parent or child tasks, or an environment can be copied, and the copy can be passed to a child task. It is also possible to get a read-only port to an environment which allows reading but not modification of the environment. Variables stored in an environment are accessible through a specific server port. There may be one read/write and one read-only port to the same environment [Thom87]. Default or empty environments can be created with system wide constants useful to all tasks (like the hostname, network-address and well known server ports) and can be copied to one another. Ports and strings can be registered in or looked up from an environment by the tasks sharing it. A task can also use more than one environment: it could have access to a widely shared "global" environment as well as its own local environment. These features play an important role in system initialization and authentication. These issues are discussed later in this chapter. For detailed description of the environment server please refer to [Gopa92].

3.2.5 The TTY Server

The TTY server regulates access to terminals and has been designed to serve any user interface in general. Although in UNIX all device related services are put inside the file system, there are certain fundamental differences between user interface devices and a disk which make it advantageous to separate the code handling user-interface from the file system. It also provides facilities for job control, process groups and special character interpretation through signaling. In order to do so, the terminal driver was modified to use a special filtering mechanism provided in the MACH device server. To facilitate I/O redirection, the interface provided by the terminal and file servers for similar operations like open, read, write, close etc have been kept as close as possible.[Gopa92]. The emulation library uses this facility to successfully implement I/O redirection.

TTY server starts the login shell. Any further reference to login server in the thesis refers to the TTY server.

3.2.6 The Network Server

The Network server provides a socket based connection oriented and datagram services using the IP suit of protocols. It runs as as a privileged kernel task with user interface through system calls. By itself, the MACH

kernel does not provide any mechanism to support inter-process communication over the network. However, the definition of MACH IPC allows for inter-process communication to be transparently extended by user level Network servers. They can act as a local representative for tasks on remote nodes. Messages destined for ports with remote receivers are sent to the local network server which forward it to the destination network server which, in turn, delivers it to the local receiver task. These user state network servers can be written over the socket facility provided by the kernel network server.[Baru92]

Of all the servers mentioned above, the environment server and the TTY server belong to the genre of generic or partially generic servers. On the other hand, the other servers are mostly designed according to the UNIX semantics. The Authentication server is implemented as a privileged thread of the process server. It may be implemented as a separate independent server. A possible extension to this set is a pipe server.

All the servers in PAWAN are statefull. They maintain some state for every task accessing the server. This becomes necessary because the users are not trusted and if no state is maintained, the server has to believe in user information and that compromises security. The state includes usually the authentication information and the server state corresponding to that task e.g. ports for communication, data structures storing some information about the user etc.

3.3 The Emulation Library

Besides doing 'emulation' and thereby providing an interface to the servers, the emulation library manages a considerable portion of process state. Thus, it displaces some of the processing required to implement various functions from the servers into the clients of these services themselves, and helps in concentrating system state in these clients. For example, it manages important pieces of information like the UNIX file descriptor table, signal mask etc. Although the user code and the emulation library code reside in the same address space, one may use the terms 'executing in user space' and 'executing in emulation space' to separately indicate the execution of the library routines [Juli92]. The emulation library is discussed in detail in the companion thesis[Das93].

3.4 General Issues

In this section we discuss some of the general issues regarding system initialization, authentication, user interface etc.

3.4.1 Public and Private Names

The concept of public and private names is implemented by the environment server. A public name is accessible to all tasks but a private name is accessible only to some chosen tasks which, in PAWAN, are the well known servers. When a user task looks up a public name, even if it is not present in its environment, it is looked up in the environments of the well known servers. On the other hand, when it looks up a private name, request is granted only if it happens to be a well known server itself.[Gopa92]

3.4.2 Server Interface to a User Task

The notion of public and private names introduced above is used to provide a secure user interface to servers. All servers have a public port and a private port which are registered in the Env server. A public port provides system wide services to all users. A private port is reserved for communication among the servers to carry out privileged opearations. The environment server's public port is available to all tasks. Any task that wishes to request some service from a server, at first initializes its environment and gets its own bootstrap port. Then it sets the bootstrap port to the Env server public port. It then obtains the public port of the server it wants to contact by requesting the Env server. After obtaining the public port of the required server, the user task directly talks to the server. The server usually returns a port, which communicates with the user task and serves all future requests through this port. Figure 3.3 describes the interaction between a user task and a server. The primary mode of communication in the system is the MACH IPC facility (MIG). It applies to communication between a user task and a server as well as communication among servers.

3.4.3 Identification of Tasks and Authentication

It follows naturally from the MACH philosophy that the identification of a user task is done by its kernel port. But unlike UNIX, where processes have a unique global pid, the task-id of a user-task is different in different servers since rights of a port are not unique to all tasks. But it does not pose a problem as long as they are appropriately translated and remain unique to a particular server so long as the server is alive.

Although for MACH applications it is possible to maintain security since ports are kernel protected capabilities, for unix applications unix-style authentication is necessary to provide protection against unauthorized and erroneous access to data. As mentioned above, the authentication server is implemented as a thread of the process server. It issues tokens of authentication and saves information about all the current tokens issued to users. Other servers ask the authentication server to verify whether an authentication token is valid [Das93]. Whenever a new task comes up in the system its

id is obtained from the file server or from its parent and a tuple is formed with the authentication token. This token is stored in a fixed memory location of the user task. The emulation library routines access this tuple and send it along with other information during each server call .

3.4.4 System Initialization

The Environment server is the first server to come up in the system. After the kernel startup, the environment server is execed by the bootload program. At first, it sets its bootstrap port to Env server public port. In the next step, it creates five tasks for Exec server, File server, Network Server, process server and TTY server respectively. It then creates the environment for each server task created above and creates public and private ports for each server, and registers them in its environment. After this, it sets the bootstrap port of each server to their env_ports. It then executes the corresponding executable files on the five tasks mentioned above by using the bootload program.

The initialization procedure done at each server is as follows :

1. my-env_port = env_init(..)
2. my_public_port = env_get_port(..)
3. my_private_port = env_get_port("..Serverprivateport",..)
4. Get public and private ports of every other server and make state for them.

make_server_state(servername, publicport, privateport);

The idea of make state is like initializing uareas for communication with other servers directly: they are supposed to use their private ports as the credential just as the ordinary user uses his kernel port.

After all the servers come up, TTY server which happens to be the last one to come up starts the login shell.

3.5 Login Shell

The login shell is spawned by the TTY server. It puts the login prompt on the terminal screen and waits for the user to type his login-name and then the password. it then checks the password by asking the file server. If the password matches, it prints the shell prompt on the screen and waits for the user to type his command. It then interprets the command and calls run() to execute the command. After the command is executed, it prints the shell prompt on the screen and waits for the next command. This goes on till the user types "exit". A possible extension of this project may be to port a standard UNIX shell like the Bourne shell.

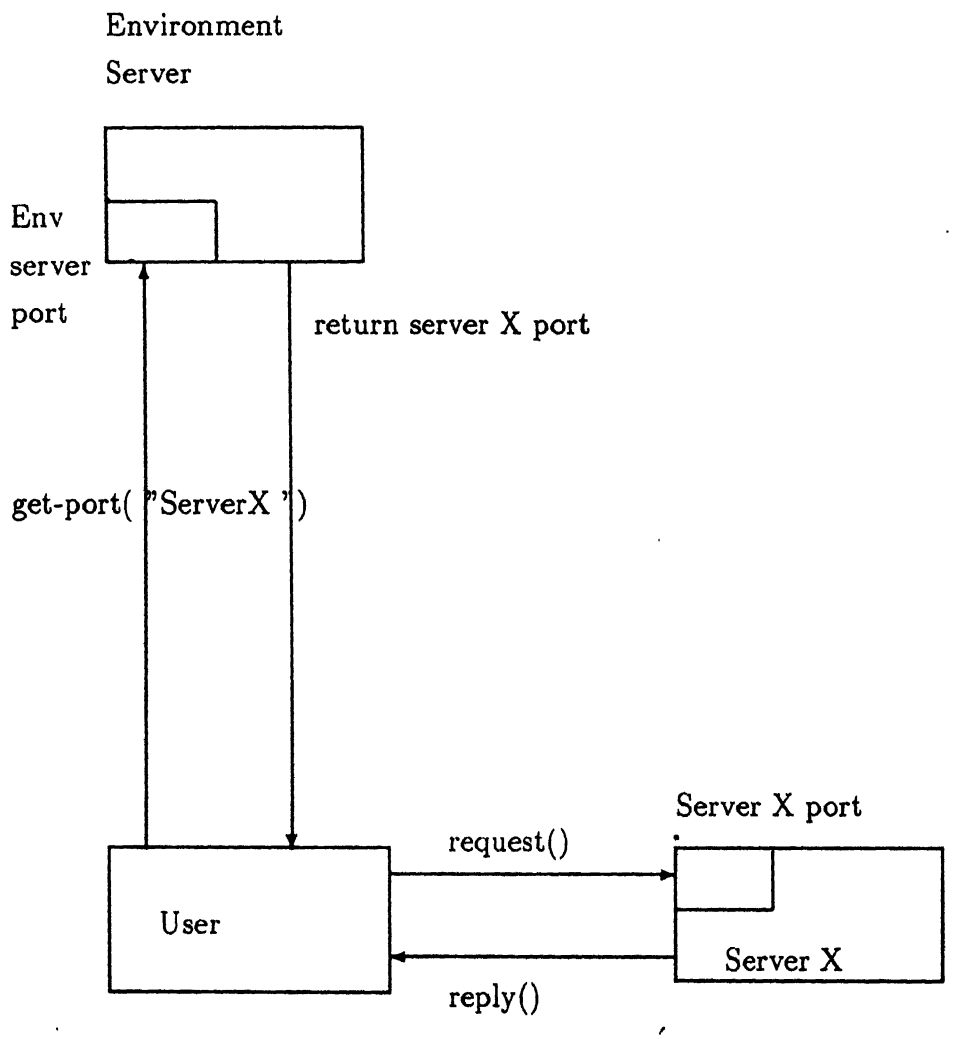


Figure 3.3: Interaction between a user and a server

Chapter 4

The File Server

4.1 Introduction

This chapter describes the file server in PAWAN. A file system compatible with the 4.3 BSD compatible file system has been successfully designed and implemented. The only file system support provided by the MACH micro kernel is a stand alone loader without any buffering. It is a read only loader and it provides services to kernel tasks only. Therefore we need a file system that fulfills the need of user tasks also and provides a service that is comparable in performance and reliability with the file systems found in conventional systems. The immediate choice was to make it compatible with the UNIX file system so that the UNIX utilities can be ported with least changes and thereby facilitating easy UNIX emulation. The file server has been written by tuning the BSD file system source code and it uses the kernel device server for device I/O. The C Threads package is extensively used to exploit the inherent parallelism offered by the MACH kernel. Our User File Server(henceforth referred to as UFS) runs as a user state server along with other servers in PAWAN and successfully supports the Unix emulation code.

4.2 Design Goals

The primary design goals of UFS are :

- To provide services identical to the Unix file system so far as possible
- To exploit MACH features viz,
  - use of threads so that a task
    can do multiple file operations simultaneously.
  - copy_on_write to minimize multiple copies of data.
- To port the BSD 4.3 source code with minimum possible modification

- To provide necessary support to the other servers.
- To run as a user task without compromising security.

4.3 Design and Implementation

UFS was originally designed and implemented by P. V. Rao[Rao92] and has been debugged and modified to meet the design goals. As stated earlier, UFS is a user state task. It has several threads of control viz, the master thread, the uarea_server_thread, the readwrite threads. UFS interacts with a user task through MIG (MACH IPC) calls. It has a public port and a private port. The private port is accessible to the other servers only and is used for privileged operations. A user task sends its requests to the public port where it is picked up by the master thread which then initiates the necessary action to serve the user request (refer to fig 4.1).

Another important aspect of UFS is the maintenance of U area. The design of U area in UFS is explained in section 4.3.1. Section 4.3.2 describes how the file server starts up and section 4.3.3 describes the working protocol. In section 4.3.4 various implementation issues are discussed and compared with those in BSD UNIX file system.

4.3.1 The design of U Area

The U area or the user area in BSD contains long term information about the state of a process. It is seen at the top of the user memory. It is a perprocess structure that may be swapped to secondary storage. This state includes

- The user and kernel mode execution states
- The state related to system calls
- The descriptor table
- The accounting information
- The resource controls
- The perprocess execution stack for the kernel[Leff89] UFS maintains the uarea not in the user memory but inside the file server itself. It is kept in a table hashed by the credential of the user tasks. This per-task U area (u.task) is created by the login server(the TTY server) when a new task is created and is updated by the file server as and when required. Sometimes UFS overwrites it completely or makes copies of it on being requested by the Exec Server. A MACH task may have several threads simultaneously requesting file system service.

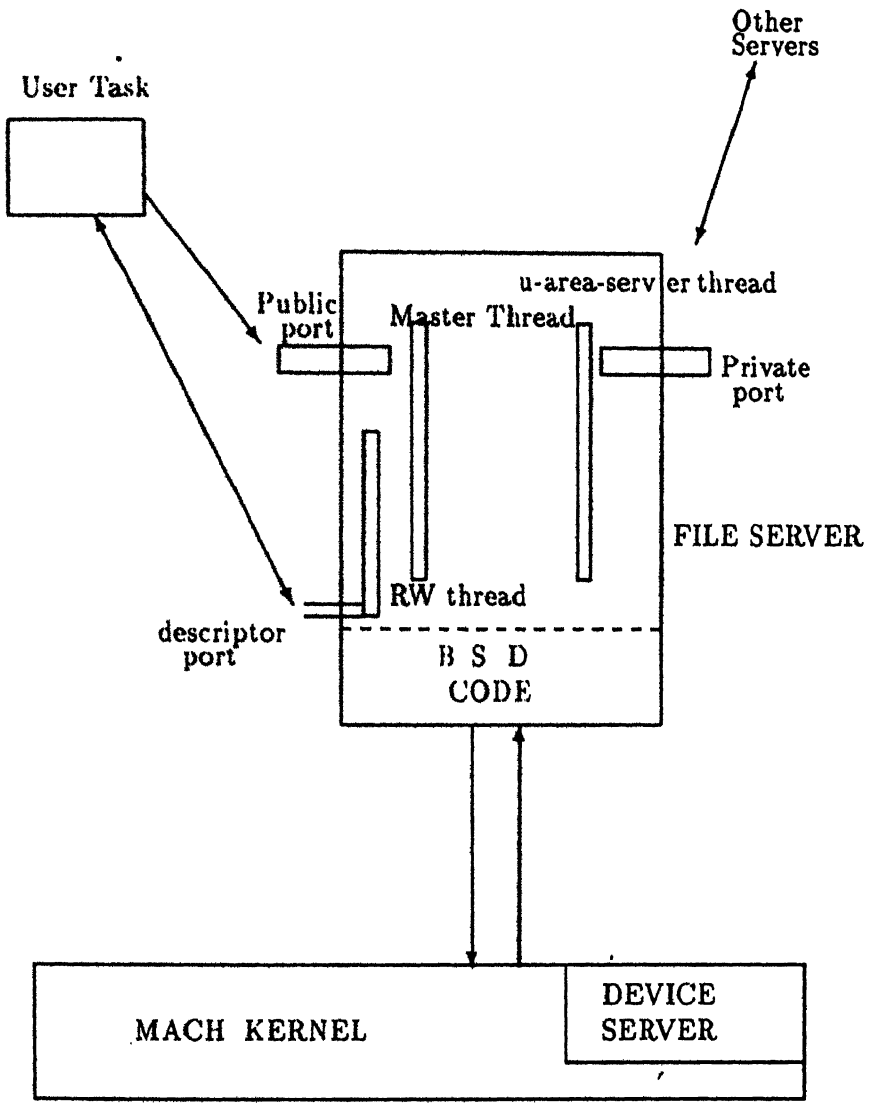


Figure 4.1: The Design of UFS

This necessitates splitting of U area into a perrequest uarea(u.request). U.task contains the fields common to all threads in a task e.g uid, gid, current directory etc. U.request is maintained in a perrequest structure called fsr and contains additional information like descriptor ports, offsets, file mapping information etc. Typically it is initialized (partly from u.task) when a file is opened and is hashed in a table by the port returned by open. Its lifetime is till the particular file is closed.

4.3.2 Authentication

So far as the issue of authentication in a system is concerned, perhaps the most crucial aspect is protection against unauthorised access of files. As discussed in chapter 3 and in the companion thesis[Das93], MACH does not provideUNIX like authentication. Therefore, without UNIX style authentication, the file server can easily be fooled to believe in a counterfeit identity of a user. Therefore, UFS works in tandem with the authentication server to protect the files of a user. It stores the uid, gid, and the authentication ticket along with the credentials i.e the kernel port of a user in u.task. Any request to UFS at the master port is accompanied by the abovementioned authentication information and UFS matches it against that stored in the u.task. If they do not match, UFS asks the authentication server to verify it. Only after being verified by the authentication server, does UFS proceed to serve the request.

4.3.3 Startup of the File Server

UFS is started by the Environment Server. At first the device server port is obtained. The first thread to come up is the master thread. It initializes the data structures e.g fsr. It then opens the root device (which is at present the disk partition sd2a in csehcl2), gets the superblock information and mounts the file system on '/'. The next step is setting up its public port, private port and the environment server port . It then makes states for other servers in UFS by creating U areas for them. Then it forks the uarea_server_thread and waits for requests from user tasks on the public port.

4.3.4 The Protocol

Whenever a new task comes up in the system, either the login server or the process server informs the file server through its private port. This information consists of the uid, ticket and the kernel port of

the new task for identification. This message is picked up by the uarea_server_thread. It creates a new u.task for the task and enters it in the uarea hash table.

When a new task wants to access UFS e.g to open a file, it sends a request to the UFS public port. This request along with other parameters for open viz file name, mode etc. also includes the uid, the kernel port of the task and the ticket for authentication. UFS retrieves the u.task from the uarea hash table and matches the ticket and uid with those stored in the u.task. If they do not match then it gets the uid and ticket for that particular task from the authentication server. If they match now the uid and ticket in u.task are updated. If they still do not match permission for carrying out the request is denied.

If permission is given, the master thread allocates a per open file request structure (fsr) as described earlier. The file is then opened and a descriptor is obtained. The operation in this part is similar to that in 4.3 BSD UNIX. The descriptor is stored in fsr. A new port is allocated and a RW (Read_Write) thread is forked which waits on this port to service all requests on this open file. The port is also stored in a table in the fsr which is indexed by the descriptor. In stead of the descriptor obtained, UFS returns this port to the user task. The new fsr is hashed with the descriptor port in a fsr_hash_table. The task uses this descriptor port for any operation to be done on the open file. The Emulation Library routines in fact hide the notion of descriptor port, uid and ticket from the user. In stead , the emulation library routine open() returns the index from the open file table where the corresponding descriptor port is stored. Figure 4.2 describes the protocol.

Child tasks inherit the files opened by its parent task and as in UNIX should be able to send requests to the same descriptor ports as its parent's. As discussed earlier, the Signal Server informs the file server about task forks and the uarea server thread creates a new u.task and replicates the uarea of the parent to that of the child. It also increases the reference counts of all the files opened by the parent task. The Exec Server copies the open file table of the parent task into the child task's memory. Thus requests to a particular open file come to the same RW thread.

A file is closed if a user requests for closing it or if the task which opened it dies. If a request for closing a file comes to the descriptor port, the RW thread retrieves the fsr structure for that file from the hash table and checks if the reference count is more than zero. If it is more than zero, the reference count is just decremented by one. Otherwise the fsr and the descriptor port are deallocated. The table of open files in u.task is also updated. If a task dies the kernel informs

the UFS about it and the uarea server thread looks up the list of descriptor ports in u.task and deallocates all of them along with the fsr structures and finally removes u.task.

4.3.5 Miscellaneous Issues and Their Comparison with BSD UNIX

- Allocation of memory:

While serving a user request, RW thread allocates memory using the system call vm_allocate( ). The requested data is copied from the buffer cache into this memory and returned to user address space through IPC. Unlike BSD UNIX where data is copied byte by byte, the virtual memory operations viz vm_allocate vm_copy etc operate on page table entries and thereby speed up copying. Data sent through IPC is shared copy on write between sender and receiver. Thus, they share the same physical pages so long as they don't write into them.

In BSD, buffers are allocated 8KB of virtual memory and 2KB of physical memory. Buffers needing more physical memory take physical pages from other free buffers[Leff89]. In UFS, all buffers are allocated 8KB of virtual memory. Physical memory is allocated by the kernel as and when pages are required[Rao92].

Interface to the Device Server :

In BSD, the block device switch(bdevsw) and character device switch (cdevsw) tables contain pointers to device driver routines for the devices in the system. In UFS, the BSD code has been modified to make remote procedure calls to the device server in stead of calling device driver routines.

- Synchronization and handling of critical code :

UFS runs as a user task. Therefore, unlike BSD, where context switches are not allowed during the execution of a system call, a RW thread may preempt another and this may lead to inconsistent global data. This is avoided by using a mutex variable at all the critical junctures. Sharing of buffers and inodes is controlled by the condition variables buf_free and ino_free. Any thread trying to use a buffer or inode checks if the corresponding condition variable is locked or not. If it is unlocked, the thread locks it and carries on. If it is locked the thread does a condition_wait. While unlocking the buffer or inode, the thread wakes up all the waiting threads by doing a condition signal.

– Passing of U area :

In BSD UNIX, any reference to a variable 'u' refers to the U area in the user's memory. In order to leave the BSD code unchanged to the extent possible, the fsr structure is passed on with every call as there are no per thread global variables [Rao92].

– Interface to UFS :

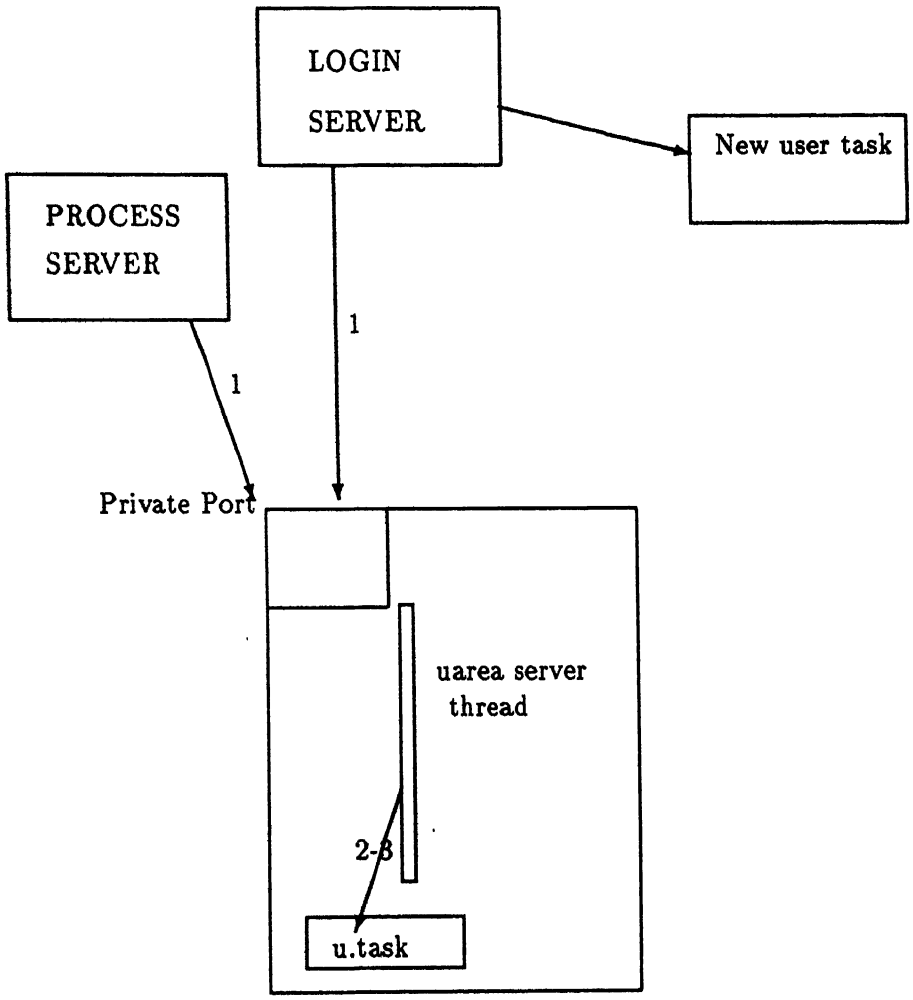
As mentioned earlier, any user task communicates with UFS through the public port. It does so by using MIG calls. The syntax of the calls are almost similar to those in UNIX. And with minimal effort, the Emulation library routines provide complete UNIX compatibility.

4.3.6 Drawbacks

In MACH, wiring of pages to prevent them from being paged out from the main memory is a privelege available only to the kernel tasks. Since UFS is a user task, its memory may be paged out if memory is used to capacity and thereby reduce the performance.

4.3.7 Conclusion

The file server has met all its design goals. Successful porting of the BSD filesystem substantiates the claim that with minimal effort one can add services at an upper layer to the MACH micro kernel. The fact that it has acheived complete UNIX compatibility is exemplified by the easy porting of programs like cat, cp etc.



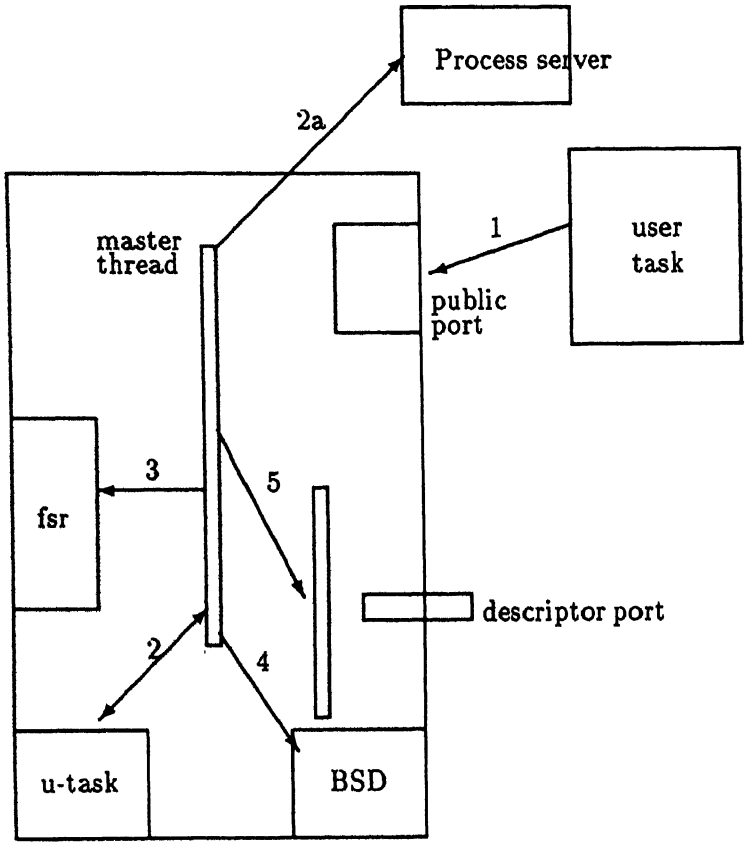
1)Login server or the Process Server informs UFS
about a new task create (uarea)
2) Create the uarea - u.task
3) Put it in the uarea-hash-table

Figure 4.2: Creation of U area

1) open(fname, mode, kernel port, ...)

2) hash the kernel port to get the u.task

2a) Authentication



3) create fsr

4) open the file

5) fork a RW thread

6) return the descriptor port

Figure 4.3: Open

1) read(descriptor port)
2) hash kernel port to get u.task
3) hash descriptor port to get the fsr
4) read
5) return data to user

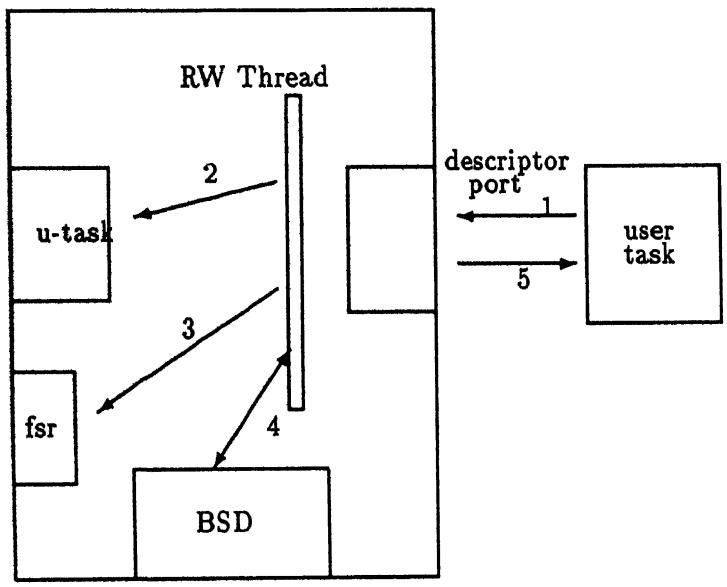
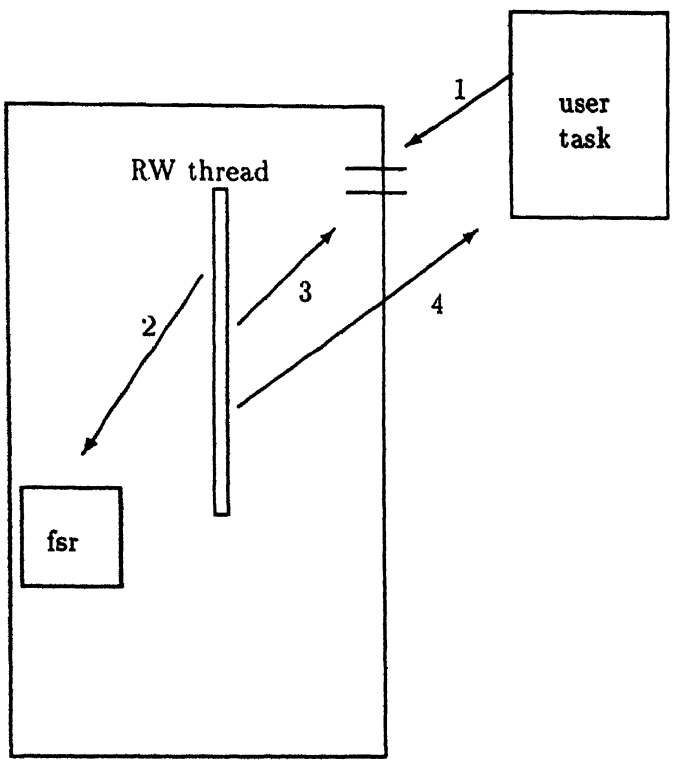


Figure 4.4: Read



1) close(fd,..)
2) clean
3) remove the port
4) rw thread sends reply to user
and destroys itself

Figure 4.5: Close

Chapter 5

Performance Evaluation

5.1 Introduction

Performance evaluation is an integral part of any operating system developement process. This is necessary for comparing the efficiency of the system with established operating systems. Not only does it help in highlighting the better qualities of the system, but more importantly, it throws light on the bottlenecks of efficiency and thereby helps in tuning the system towards better performance.

We examined the performance of our multiserver UNIX emulation system through the use of system oriented benchmarks. And it has been found that the performance of PAWAN is poor compared to that of UNIX. It must be stressed here that the primary goal of the design and implementation of PAWAN was to successfully build an emulation system using the multiserver model. Although efficiency was a factor that helped in deciding the design. for the present implementation, it never played a major role and it was not expected to give performance comparable to that of UNIX or the Single Server Model. But nevertheless that alone does not account for its performance. A natural conclusion is that efficiency issues in PAWAN need serious consideration in future.

A detailed description of the tests done is given in the next section and the results are compared with those of UNIX and explanations are given along with suggestions for improvements.

5.2 Description of the Experiments

PAWAN does not have a sophisticated user interface. That makes the task of doing performance tests slightly tedious. Most of the tests

Operation	UNIX	PAWAN	PAWAN(inside UFS)	Single Server Model
lseek	100micros	10ms	80micros	170micros
lseek+read 1b	0.67ms	27ms	7ms	.35ms
lseek+read 1k	0.90ms	27ms	8ms	-
lseek+read 2k	1ms	29ms	8ms	-
lseek+read 4k	1.2ms	31ms	9ms	-
lseek+read 8k	2ms	34ms	12	5.5ms

Table 5.1: OPERATING COST OF read() and lseek()

which have been carried out are chosen in such a way that they reflect system performance in general. But since only a few standard benchmark programs, that could be run on PAWAN, were available, the tests were limited to finding out the time taken in the most basic and frequent operations in UNIX and their equivalents in PAWAN.

5.2.1 Timing

The system clock in HORIZON III has a frequency of 60Hz i.e. each clock tick is after a duration of 16ms. This frequency is not fine enough to measure the time for operations like read, write, lseek, getpid etc. But due to the unavailability of a counter, efforts to get more accurate timing was abandoned. Instead, the time taken to carry out a large number of the same operation(usually around a hundred times) was measured and the average was found out. Since the time taken depends on the load on the system also these sequence of operations were carefully carried out repeatedly to avoid erroneous collection of data.

5.2.2 FILE I/O

One of the major costs in any UNIX system is the cost of data movement to and from the disk subsystem. And among the system calls related to such operations, the ones which dominate are read, write, lseek, close and stat. The performance of the multi threaded Unix server that was developed at CMU, was also available to us [Dean90]. The values obtained for different sizes of data copying is shown in table 5.1.

In the above operations, lseek and read are taken together because hundreds of read() are done repeatedly to find the average time and for larger sizes of data transfer, more than one block has to be fetched. This may lead to an extra overhead and therefore the same block is

Operation	UNIX	PAWAN	PAWAN(inside UFS)
open	16ms	32ms	18ms
open read 8K	50ms	92ms	70ms
open read 8K close	60ms	112ms	85ms

Table 5.2: open(), read() and close()

read over and over again by using lseek. A graph drawn from the above table is shown in figure 5.1.

As mentioned in Chapter 3, one of the reasons for choosing the multiserver model is that it is suitable for the LAN based environment in our laboratory. But, as of now, PAWAN runs only in HORIZON III. Therefore, degradation of performance in comparison with the single server model or UNIX is not unexpected. But as the values in table 1 show, the time taken for read() is too high for PAWAN. One of the reasons for this is the overhead involved in MIG. Since Mach is a message passing system, most of the interaction between a user and a server or between a server and the kernel is done through RPC. Typically, a read() involves two MIG calls one from user to the file server and another from file server to the device server. The approximate time taken by a MIG call is on the average around 10ms. Therefore, irrespective of the methods used for minimizing the copies of data, the MIG overhead nullifies the advantages gained from this. The time taken for read() inside UFS shows a marked improvement over that in a read() from a user task. It may be pointed out here that since lseek inside UFS and lseek in UNIX takes negligible time compared to read(), the comparison of these two sets of values gives us a clear picture of the costs involved in read(). But the MIG overhead alone does not explain the performance. The difference between the performance of UFS and UNIX can be explained by the extra code in UFS read() and the vm_allocate() system call which is called every time read is called. Moreover, the code written is also not optimal. Besides these, there might be inefficiencies inside the kernel in task switching etc which contribute to deteriorating performance.

The block size in UFS is 8K. Table 5.2 shows the time taken for opening a file, reading 8K from it and closing the file. This read() involves getting a new block from the disk. Here again, the performance can be explained by the reasons mentioned above. The overhead involved in open and close operations are discussed in Chapter 4.

Time taken for creation of a file and writing data of various size is

Operation	UNIX	PAWAN
create	50ms	72ms
create write 0b	75ms	96ms
create write 8k	90ms	160ms

Table 5.3: TIME FOR create() AND write()

Operating System	mkdir()
Horizon III UNIX	200ms
PAWAN	1.2s
MACH Release 2.5	4sec
MACH Single Server	1sec

Table 5.4: DIRECTORY CREATION

shown in table 5.3.

The performance data of the multithreaded single server model have been taken using the benchmarks developed for the Andrew file system. This benchmark stresses directory and file creation etc. The values for this system are referred from [Dean90]. They were taken on a Sun 3/60 machine. As shown in table 5.4, in operations such as "mkdir", "write" etc, where the time taken is high and the MIG overhead is negligible compared to it, the time taken in PAWAN is comparable to the multihreaded Unix server.

5.2.3 Process Management

Standard UNIX benchmarks use the system call getpid() to find the time involved in a typical system call. The getpid() call in PAWAN is implemented entirely in server level. Therefore, to find out the overheads involved in a system call like getpid() in UNIX, we added a new dummy system call to the MACH kernel. Table 5 shows some of the standard process management costs.

Operation	UNIX	PAWAN
getpid	90micros	50micros
fork	16.6ms	398ms
exec	66.67ms	840ms

Table 5.5: PROCESS MANAGEMENT COSTS

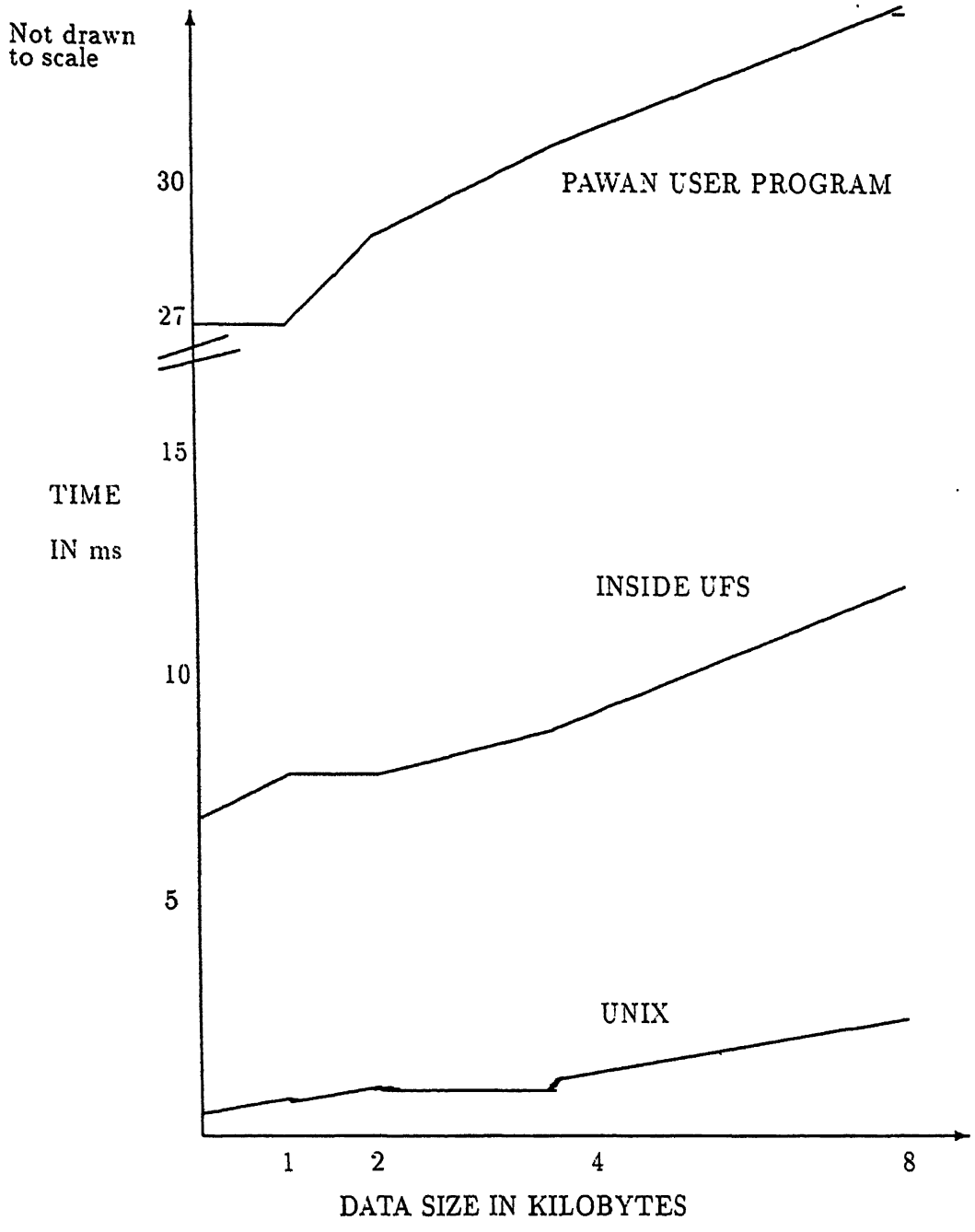


Figure 5.1: cost of read() and lseek()

Chapter 6

Conclusion

The work described in this, and the companion theses, exhibits the successful implementation of a Mach-based operating system which emulates UNIX. As stressed in the earlier chapters, the major motivation and goal behind this thesis has been to construct an operating system as a set of system servers which sit on top of a minimal Microkernel and support the implementation of functions that are specific to a particular operating system. This approach promises to meet system builders' need for a sophisticated operating system development environment that can cope with growing complexity, new architectures, and changing market conditions. And the fact that PAWAN has met its design goals further substantiates the claim that this approach is here to stay.

The very idea of microkernel based operating systems is rooted in the belief that in order to make way for configurability, potential inefficiency in certain cases is not unwelcome. And in our case, efficiency has never been a major goal. As discussed in the previous chapter, the efficiency issues have to be treated seriously to make PAWAN a real success. Therefore, tuning up of PAWAN to achieve better efficiency may be a possible extension to this project.

During the implementation of this emulation system, the file server was tuned to provide UNIX like file system support and at the same time exploit the MACH features. While porting the standard utilities, quite a few bugs in the file server and the emulation library surfaced and it helped making the UFS more robust.

Any research in operating systems, more particularly in distributed systems, is presently influenced to a large extent by attempts to stay compatible with UNIX. Mach is no exception to this philosophy. Although PAWAN successfully emulates UNIX, it is not yet a full fledged

operating system. Enrichment of the emulation library and porting of standard utilities is another possible extension to this project although this might turn out to be a never ending process. The most important among them being an editor and the gcc compiler. At present, programs have to be edited and compiled on the SUN workstations. It may be pointed out here that porting of the Bourne Again Shell was one of the goals set in the beginning. But it never materialized owing to lack of time. The login shell which is started by the TTY Server may be written as an independent server.

The potential benefits of micro-kernel support for real-time systems has been discussed earlier. PAWAN can provide a platform to explore this area not only because it is based on a micro-kernel but also because of the UNIX compatibility it provides.

Bibliography

[Abro89]Abrossimov,V., Rozier,M., Gien,M., *Virtual Memory Management in CHORUS*, LNCS, Vol 433, Springer-Verlag, 1989.

[Acce86] Accetta,M., Baron,R., Golub,D., Rashid,R., Tevanian,A., and Young,M., *MACH: A New Kernel Foundation for UNIX Development*, Technical Report, School of Computer Science, Carnegie Mellon University, August 1986.

[Ashi92] Ashish,S., *PAWAN: A MACH based UNIX system -I*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Bach86] Bach,M.J., *The Design of the UNIX Operating System*, Prentice-Hall Inc., Engelwood Cliffs, May 1986.

[Baro88] Baron,R.V., Black,D., Bolosky,W., Chew,J., Draves,R.P., Golub,D.B., Rashid,R.F., Tevanian,A.Jr., and Young,M.W., *MACH Kernel Interface Manual, Online MACH Documents (unpublished)*, School of Computer Science, Carnegie Mellon University, October 1988.

[Baru92] Baruah,M., *PAWAN : A MACH based UNIX System(III)*, M. Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Blac89] Black,D.L., *Mach Interface Proposals - Priorities, Handoff, Wiring*. August 1989. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University.

[Bolo87] Bolosky,W., Rashid,R., Tevanian,A.Jr., Young,M., Golub,D., Baron,R., Black,D., and Chew,J., *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures*, Technical Report CMU-CS-87-140, School of Computer Science, Carnegie Mellon University, July 1987.

[Cher84] Cheriton,D.R., *The V-Kernel: A Software Base for Dis-*

tributed Systems, IEEE Software, Vol 1 no.2, pp. 77-107.

[Cher88] Cheriton,D.R., *The V Distributed System*, Communications of the ACM, 31(3), March 1988.

[Coop88] Cooper,E.C., and Draves,R.P., *C Threads*, Technical Report CMU- CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[Coul88] Couloris, G.F., Dollimore,J., *Distributed Systems- Concepts and Design*, Addison-Wesley, July 1988.

[Das93] Das,J.J.H., *UNIX Emulation Services in PAWAN(I)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1993.

[Dean90] Dean,R., Golub,D., Forin,A., Rashid,R., *UNIX as an Application Program*, Proceedings of the 1990 Summer Usenix, Usenix, June 1990.

[Drav89] Draves,R.P., Jones,M.B., and Thompson,M.R., *MIG - The MACH Interface Generator*, Online MACH Documents (unpublished), School of Computer Science, Carnegie Mellon University, July 1989.

[Gien91] Gien,M., *Next Generation Operating System Architecture*, LNCS, Vol. 563, July 1991.

[Gold89] Goldstien,I., *Introduction to th MACH Development workshop*, Proceedings of the Workshop on MACH Development, Boston, 1989.

[Golu87] Golub,D.B., Tevanian,A.Jr., Rashid,R., Black,D.L., Cooper,E., and Young,M.W., *MACH Threads and the UNIX Kernel: The Battle for Control*, Technical Report CMU-CS-87-149, School of Computer Science, Carnegie Mellon University, August 1987.

[Gopa92] Gopal,B., *PAWAN : A MACH based UNIX System(II)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Jone86] Jones,M.B., Rashid,R.T., *MACH and Matchmaker : Kernel and Language Support for Object Oriented Distributed systems*, ACM Sigplan Notices, Vol. 21, no.11, pp. 67-77.

[Juli92] Julin,D.P., Chew, J.J., Stevenson, J.M., Guedes, P., Neves, P., Roy, P., *Generalised Emulation Services for MACH3.0: Overview,*

Experiences and Current status, MACH Symposium, USENIX Association.

[Leff89] Leffler,S.J., McKusick,M.J., Karels,M.J., Quarterman,J.S., *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley Publishing Co., May 1989.

[Mull85] Mullender, S,J., Tanenbaum, A.S., *A distributed File Server Based on Optimistic Concurrency Control*, Proceedings of the Tenth ACM Symposium on Operating Systems, Dec. 1985.

[Pete90] Peterson, L.M., Hutchinson, N.C., Malley, S.M., and Rao, H.C., *The X-kernel: A Platform for Accessing Internet Resources*, IEEE Computer, 23(5), May 1990.

[Rao92] Rao, P.V., *PAWAN : A MACH based UNIX System(IV)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Rash86] Rashid,R.F., *Threads of a New System*, UNIX Review August 1986.

[Rash87] Rashid,R.F., *Designs for Parallel Architectures* UNIX Review, April 1987.

[Rene89] Renesser, R.V., Tanenbaum, A.S., *The Evolution of a Distributed Operating System*, LNCS, Vol 433, Springer-Verlag.

[Tane90] Tanenbaum et al., *Amoeba - A Distributed Operating System for the 1990*, IEEE Computer, Vol. 18 No. 2, 1990.

[Teva87] Tevanian,A. Jr., and Rashid, R., *MACH: A Basis for Future UNIX Development*, Technical Report CMU-CS-87-139, School of Computer Science, Carnegie Mellon University, Pittsburgh, June 1987.

[Thom87] Thompson,M.R., Tevanian,A.Jr., Rashid,R., Young,M.W., Golub,D.B., Bolosky,W., and Sanzi,R., *A UNIX Interface for Shared Memory and Memory Mapped Files Under MACH*, Technical Report, School of Computer Science, Carnegie Mellon University, Pittsburgh, July 1987.

[Walk83] Walker et al., *The LOCUS Distributed Operating System*, Proceedings of the Ninth Symposium on Operating Systems Princi-

ples, October 1983.

[Youn87] Young,M., Tevanian,A.Jr., Rashid,R., Golub,D., Eppinger,J., Chew,J., Bolosky,W., Black,D., and Baron,R., *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*, Proceedings of the Eleventh ACM Symposium on Operating System Principles, Austin, Texas 1987.